SPECTRAL METHODS FOR PLA DECOMPOSITION AND FAULT DETECTION

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

By
KRUTIBAS BISWAL

to the
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
February, 1993

To

my

family members

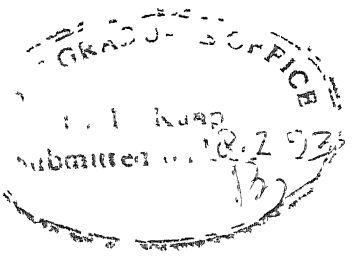
ACKNOWLEDGEMENT

I am grately indebted to *Dr M M Hasan*, my thesis superviser for his valuable suggestion, motivation and spirited guidance throughout the course of this work Beyond very general suggestions and hints, he let me do my own thinking as far as possible It was thus that my thesis work itself became an enriching experience

I acknowledge the worm and cheerful company of Barada, Himanshu, Dandapat Sir, and the inspiring suggestions and helps of Ashok Bhai at all stages of this work

I take this opportunity to thank all my friends of Hall-V, who made my stay at IIT-K a memorable one

Last but not the least, my heartfelt thanks to my family members for making me what I am today



CERTIFICATE

Certified that the work contained in the thesis entitled, 'SPECTRAL METHODS FOR PLA DECOMPOSITION AND FAULT DETECTION,' by Mr KRUTIBAS BISWAL, has been done under my supervision and the same has not been submitted elsewhere for a degree

February, 1993

Dr M M Hasan
Professor
Department of Electrical Engineering
I I T KANPUR

Abstract

A theory has been developed to calculate the Hadamard-Walsh transform from a list of cubes specification of incompletely specified Boolean functions An efficient algorithm to generate disjoint cubes from non-disjoint ones of a cover has been developed The transformation algorithm makes use of properties of list of disjoint cubes and allows the determination of the spectral coefficients in an independent way

Decomposition of Boolean function at the functional level is a difficult problem since it requires a global approach We discuss linear decompositions of Boolean functions implemented as Programmable Logic Arrays (PLAs) Results with experimentation with certain functions shows that functions with imbedded addition are most benefited from this approach

Syndrome testing is particularly useful in two level circuits such as PLAs It is shown that weighted sum of syndromes of all the outputs covers all single stuck-at-faults, bridging faults and cross-point faults An algorithm is presented, which checks for the WSS testability of a PLA and suggests hadware modifications, if found WSS-untestable

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

During the past three decades a new field of digital logic theory has been born and its boundaries continuously extended There was an eager attempt to apply orthogonal transform techniques to the design, optimization, and testing of digital logic It was conjectured that these so called spectral methods would provide a unified approach to the synthesis of anlog and digital (binary and multivalued) circuits, making it possible to apply the powerful Fourier transform techniques, widely used in analog circuit synthesis, to digital network design Two motives for this work may be discerned first, the purely academic one of applying existing mathematical techniques to the field of digital logic design, and second, the increasing appreciation of the limitations of existing algebraic and geometric methods in handling digital data for logic network design purposes The theory that subsequently emerged has proved to be fascinating

Digital logic design is a peculiar discipline in that the logic design for a given requirement, which may be in total a very large system, is frequently made with the use of sophisticated design procedure Very great sophistication, however, may be present in the logical verification of the complete assembly and in its translation into a microelectronic realisation Though the increasing capabilities of LSI and VLSI fabrication have so far kept pace with the designers' need, there has always been a rising demand for "better" design techniques

However, the spectral methods of digital logic theory represents a modern approach to expressing conventional digital data, one which can provide various insights into the structure of the data which are absent from the classical Boolean algebra and truth-table formats The pioneering work, particularly that of Karpovsky and Lechner, forms the basis of this approach Others have contributed and amplified the basic concepts and have translated the under lying mathematics into engineering tools which may be more acceptable

to a digital logic designer

Spectral techniques have been applied to Boolean function classification [1], [2], [3], [4], disjoint decomposition [2], [5], parallel and serial linear decomposition [2], [6], [7], [8], [5], multiplexer synthesis [2], [9], prime implicant extraction by spectral summation [2], [8], [10], threshold logic synthesis [1], [8] and state assignment [7], [11] Spectral methods for testing of logical networks by verification of the coefficients in the spectrum have been developed [2], [6],[8], [12], [13], [14], [15], [16], [17], [18], [19] It should be stressed that an important problem of finding the complement of a Boolean function that has high complexity in the Boolean domain [20], [21] can be solved very easily in the spectral domain because complementing the Boolean function corresponds to changing the sign of every spectral coefficient Tautology of a Boolean function can be verified by calculating a certain coefficient (DC coefficient) The renewed interest in applications of spectral methods in logic synthesis is caused by their excellent design for testability properties and the possibility of performing the decomposition with gates other than the ones used in most classical approaches Spectral techniques have also been used for data transmission, digital filtering, image processing and pattern analysis This thesis will attempt to introduce the underlying theory of this area, that of orthogonal transforms and resulting spectral data While the underlying theory of this area is applicable to any-valued logic, and not exclusively to the binary case, we will confine our discussion herewith to two-valued digital netwotks

The second chapter gives a complete explanation of the spectra of discrete functions and their calculations An efficient algorithm for calculation of Hadamard–Walsh transform from a cube array specification of incompletely specified Boolean functions has been designed

In the third chapter we discuss the linear decomposition of Boolean functions at the functional level with emphasis to Programmable Logic Arrays (PLAs) An procedure for the linear decompositon is presented. of Boolean functions Experimental results are presented for some commonly used functions implemented as PLAs which establish that such decomposition can often result in imroved implementation of logic functions

The most recent application for spectral techniques has been in fault diagnosis This exciting area is dealt with in the fourth chapter One of the serious problems with digital circuits concerns their testing There are two aspects to this – first to verify whether or not a circuit is performing correctly and, second, if there is a fault, to find its location

(fault isolation). Of course, from the user's point of view there is not usually much purpose in locating a fault beyond identifying a chip that needs replacing There is no need to identify the the exact location of the fault in the chip We shall discuss about a simple and effective fault detection technique known in the literature as *Syndrome testing* We apply this technique with the algorithms discussed in second chapter to fault detection of PLAs

All the algorithms discussed in the second, third and fourth chapters are suitable for automation and have been implemented in 'C' language

Chapter 2

THE SPECTRA OF DISCRETE FUNCTIONS

In this chapter we shall review the theoretical aspects of spectra of discrete functions in general and later on formulate an algorithm to calculate the spectra of incompletely specified Boolean functions represented either as a list of true/false minterms or as an already minimized sum-of-products Boolean expression (SOPE)

2.1 Boolean and Spectral domains

The majority of existing methods for design and analysis of switching circuits are concerned with the properties of Boolean functions since it was proved that the Boolean domain provides a precise model for the analysis of switching circuits The behaviour of a device is represented by a function $\mathcal{F}(x_1, x_2, \quad, x_n)$ of its input variables x_1,x_2, $\quad$,x_n This function can most conveniently be defined by a truth table For example, a function $\mathcal{F}(x_1, x_2, x_3, x_4)$ of four variables x_1, x_2, x_3 and x_4 is illustrated in Fig 2 1

It is common to use the product and sum operators of the Boolean algebra together with negation to define such functions–for example, $\mathcal{F}(x_1, x_2) = x_1\vec{x}_2 + \vec{x}_1 x_2$ Part of the problem with the definition in the Boolean domain is that each of the entries in the column for $\mathcal{F}(x_1, x_2, x_3, x_4)$ in Fig 2 1 tells us precisely the behaviour of the function at a single point but nothing of its behaviour at other points It is possible to give an alternate representation of a function where the information about the function is much more global in nature This alternate representation is in the spectral domain, and it has been demonstrated [2],[7] that a number of properties are much easily deduced in the spectral domain than in the Boolean one

The basic idea of the spectral domain, and how to get there, is illustrated in Fig 2 2 If we are to avoid losing information, we shall have to ensure that the transform can be reversed, *i e* , that we can move to and from the spectral domain without any loss of

x_4	x_3	x_2	x_1	$\mathcal{F}(x_1, x_2, x_3, x_4)$
0	0	0	0	-
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 2 1 The function $\mathcal{F}(x_1, x_2, x_3, x_4)$

information

The question arises as to the reason for considering the spectral domain and if there are any real purposes for its use To understand the first difference between the Boolean and spectral domain, let us consider a Boolean function $\mathcal{F}(X)$ of n variables One row of the table defining this function provides complete and precise information about the behaviour of the function for one combination of the input variables Of course, it does not tell us anything about the value of the function anywhere else The combination of the knowledge of the behaviour of $\mathcal{F}(X)$ for the 2^n rows of the table gives a complete definition of the functions Similarly the spectrum for a function of n variables also contain 2^n values, which together completely define the function and can be used to recover its

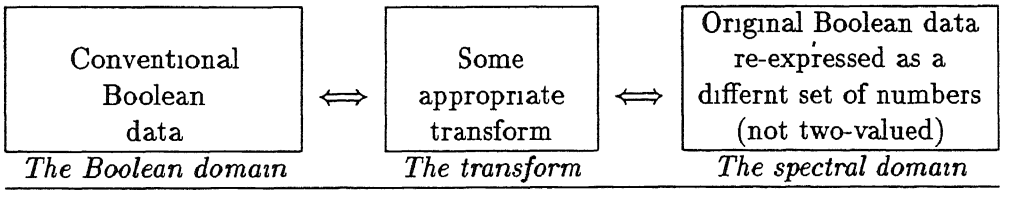


Figure 2 2 The transform

Boolean specification Each of the 2^n values in the spectrum (the spectral coefficients) contains some information about the behaviour of the function at all 2^n points, but does not contain complete information about any of them The combination of all the values in the spectrum does lead to complete information about the function, but each individual coefficient gives us some global information about the whole function

2.2 The Transformation of Binary Data

As shown in Fig 2 1 any given combinatorial function $\mathcal{F}(X)$ may be explicitly defined by its truth table, which lists all 2^n combinations, against each combination being given the local function output value $\mathcal{F}(X)$ Conventionally, all x_i input variables and the output value of $\mathcal{F}(X)$ are expressed in {0,1} notation Note that each input combination and the corresponding output value of $\mathcal{F}(X)$ is discrete data, no information about the output value of $\mathcal{F}(X)$ at any input combination being obtainable from any other input combination

With the order of tabulation of the input variables standardised in straight binary code (SBC), the 2^n local output values may be treated as a column vector **Z** defining $\mathcal{F}(X)$ such that the conventional $\{ 0, 1, - \}$ values (false, true and don't care minterms) correspond to $\{ 0, 1, 0\,5 \}$ codings respectively as in Fig 2 3(a) Should the binary logic values be recoded from $\{ 0, 1, - \}$ to $\{ +1, -1, 0 \}$, respectively, then an alternative column vector **Y** defining $\mathcal{F}(X)$ is obtained, as shown in Fig 2 3(b) The reason for such recoding can be found in [2], [7]

In both **Y** and **Z** we have 2^n entries necessary to fully define an n-variable function $\mathcal{F}(X)$. As an altrnative to these 2^n entries, we may also specify 2^n coefficients that define $\mathcal{F}(X)$, these coefficients being the *spectral coefficients* of $\mathcal{F}(X)$, these coefficients may also be listed in a defined order to give a column vector representation, which is identified by **R** or **S** As will be seen, the entries in **R** and **S** will no longer be binary values as in the local value truth table vectors **Y** and **Z**

The definition of the individual spectral coefficients in **R** and **S** for any $\mathcal{F}(X)$ may be approached from several mathematical viewpoints Here we shall first consider the Hadamard transform $\mathbf{T}^n$, which will directly generate **S** from **Y** or **R** from **Z**, following which we consider alternative but equivalent mathematical definitions and aspects

(a) In $\{0, 1\}$ notation, function vector **Z**

	Truthtable					Output vector	
Minterm	x_4	x_3	x_2	x_1	$\mathcal{F}(X)$		**Z**
m_0	0	0	0	0	0 5	z_0	0 5
m_1	0	0	0	1	1	z_1	1
m_2	0	0	1	0	1	z_2	1
m_3	0	0	1	1	1	z_3	1
m_4	0	1	0	0	0	z_4	0
m_5	0	1	0	1	1	z_5	1
m_6	0	1	1	0	1	z_6	1
m_7	0	1	1	1	1	z_7	1
m_8	1	0	0	0	0	z_8	0
m_9	1	0	0	1	0	z_9	0
m_{10}	1	0	1	0	1	z_{10}	1
m_{11}	1	0	1	1	1	z_{11}	1
m_{12}	1	1	0	0	1	z_{12}	1
m_{13}	1	1	0	1	1	z_{13}	1
m_{14}	1	1	1	0	1	z_{14}	1
m_{15}	1	1	1	1	1	z_{15}	1

(b) In $\{+1, -1\}$ notation, function vector **Y**

	Truthtable(recoded)					Output vector	
Minterm	x_4	x_3	x_2	x_1	$\mathcal{F}(X)$		**Y**
m_0	1	1	1	1	0	y_0	0
m_1	1	1	1	-1	-1	y_1	-1
m_2	1	1	-1	1	-1	y_2	-1
m_3	1	1	-1	-1	-1	y_3	-1
m_4	1	-1	1	1	1	y_4	1
m_5	1	-1	1	-1	-1	y_5	-1
m_6	1	-1	-1	1	-1	y_6	-1
m_7	1	-1	-1	-1	-1	y_7	-1
m_8	-1	1	1	1	1	y_8	1
m_9	-1	1	1	-1	1	y_9	1
m_{10}	-1	1	-1	1	-1	y_{10}	-1
m_{11}	-1	1	-1	-1	-1	y_{11}	-1
m_{12}	-1	-1	1	1	-1	y_{12}	-1
m_{13}	-1	-1	1	-1	-1	y_{13}	-1
m_{14}	-1	-1	-1	1	-1	y_{14}	-1
m_{15}	-1	-1	-1	-1	-1	y_{15}	-1

Figure 2 3 Truthtable formats for example four-variable function

2.2.1 The Hadamard Orthogonal Transform Matrix

The Hadamard transform is a complete orthogonal square matrix, with row and column entries $\in \{+1,-1\}$, and with a recursive structure as follows

$$\mathbf{T}^n = \begin{bmatrix} \mathbf{T}^{n-1} & \mathbf{T}^{n-1} \\ \mathbf{T}^{n-1} & -\mathbf{T}^{n-1} \end{bmatrix} \tag{2 1}$$

Note that the dimensions of the transform are $2^n X 2^n$ for any n For increasing n we have the real integer values

$$\begin{aligned} \mathbf{T}^0 &= +1 \\ \mathbf{T}^1 &= \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad (2\ 2) \\ \mathbf{T}^2 &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, \ etc \end{aligned}$$

We may alternatively express this recursive structure by

$$\begin{aligned} \mathbf{T}^n &= \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \mathbf{T}^{n-1} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes^n \qquad (2\ 3) \end{aligned}$$

where $\otimes$ here denotes the Kronecker product operator

The value of each element t_{jk} in the Hadamard matrix may be individually defined by mathematically summing the element-by-element multiplication of the binary (mod 2) expansions of j and k (call this v) and taking the vth power of -1 For example, taking the third row ($j = 10$), fourth column ($k = 11$) entry t_{23} of $\mathbf{T}^n$, shown below we have

$$\begin{array}{c} k(binary)\ 00\ \ 01\ \ 10\ \ 11 \\ j(binary) \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \end{array}$$

$$j = 10, k = 11, \Rightarrow v = \{1\ 1 + 0\ 1\} = 1$$

$$\Rightarrow t^{23} = (-1)^1 = -1$$

This may formally be expressed by

$$t_{jk} = (-1)^{\sum_{n=0}^{n=1} j_n k_n} \tag{2 4}$$

where j_n,k_n are determined by the binary expansions of j,k respectively, j,$k \in 0$ to 2^n-1, and where

$$j = \{j_{n-1}2^{n-1} + j_{n-2}2^{n-2} + \quad + j_0 2^0\}$$

$$k = \{k_{n-1}2^{n-1} + k_{n-2}2^{n-2} + \quad + k_0 2^0\}$$

The individual row vectors $\mathbf{T}_{j*}$ of $\mathbf{T}^n$ define a complete orthogonal series of functions, which is closed under row multiplication The inner product of any two matrix rows j_α,j_β has the orthogonal property

$$\begin{aligned} \sum_{k=0}^{2^n-1} t_{j_\alpha k} t_{j_\beta k} &= 2^n \; if \; \alpha = \beta \\ &= 0 \; otherwise \end{aligned} \tag{2 5}$$

This inner product relationship holds also for any two-column vectors $\mathbf{T}_{*k}$

The Hadamard transform matrix has the additional property that its transpose is identical to itself, that is, $\mathbf{T}^n = [\mathbf{T}^n]^t$, thus giving the simple inverse transform relationship for any n of

$$[\mathbf{T}^n]^{-1} = \frac{1}{2^n}[\mathbf{T}^n] \tag{2 6}$$

2.2.2 The Hadamard Transformation of Binary Data

The Hadamard matrix may be used to transform any 2^n column vector of numbers into an alternative data column vector No information is lost in this transformation, the original-data being fully recoverable from the resulting vector by application of the inverse transform $[\mathbf{T}^n]^{-1}$ We will use it in the transformation of a column vector $\mathbf{Y}$ or $\mathbf{Z}$ representing the output truthtable of a given function $\mathcal{F}(X)$

Example Considering the four-variable example function of Fig 2 3 and taking the (conventional) 0, 1 coding we may transform $\mathbf{Z}$ with an $n = 4$ Hadamard transform as

follows

$$\mathbf{T}^4 = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1
\end{bmatrix}$$

$$\mathbf{Z}^t = \begin{bmatrix} 0\,5 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Now,

$$\mathbf{R}^t = [\mathbf{T}^4\mathbf{Z}]^t \tag{2 7}$$

$$= \begin{bmatrix} 12\,5 & -1\,5 & -3\,5 & -1\,5 & -1\,5 & 0\,5 & -1\,5 & 0\,5 \\ 0\,5 & -1\,5 & 0\,5 & -1\,5 & 2\,5 & 0\,5 & 2\,5 & 0\,5 \end{bmatrix} \tag{2 8}$$

If the output vector of $\mathcal{F}(X)$ is recoded from $\{0, 1, -\}$ to $\{+1, -1, 0\}$ then we have,

$$\mathbf{Y}^t = \begin{bmatrix} 0 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

$$\mathbf{S}^t = [\mathbf{T}^4\mathbf{Y}]^t \tag{2 9}$$

$$= \begin{bmatrix} -9 & 3 & 7 & 3 & 3 & -1 & 3 & -1 & -1 & 3 & -1 & 3 & -5 & -1 & -5 & -1 \end{bmatrix} \tag{2 10}$$

The resulting vectors spectra $\mathbf{R}$ and $\mathbf{S}$ each uniquely represent the given function $\mathcal{F}(X)$ The individual coefficient values in $\mathbf{R}$ and $\mathbf{S}$ are linearly related due to the linear recoding between $\mathbf{Z}$ and $\mathbf{Y}$, the relationships being

$$y_j = 1 - 2z_j \tag{2 11}$$

$$r_0 = \frac{1}{2}(2^n - s_0) \tag{2 12}$$

$$r_\alpha = -\frac{1}{2}s_\alpha,\ \alpha \neq 0 \tag{2 13}$$

The inverse transformation back from the spectraldomain to the Boolean domain is directly given by,

$$\begin{aligned} \mathbf{Z} &= [\mathbf{T}^n]^{-1}\mathbf{R} \\ &= \frac{1}{2^n}[\mathbf{T}^n]^{-1}\mathbf{R} \end{aligned} \tag{2 14}$$

$$\begin{aligned} \mathbf{Y} &= [\mathbf{T}^n]^{-1}\mathbf{S} \\ &= \frac{1}{2^n}[\mathbf{T}^n]^{-1}\mathbf{S} \end{aligned} \tag{2 15}$$

Whilst the coefficients $\mathbf{R}$ or $\mathbf{S}$ of equations (2 8) or (2 10) uniquely represent a single combinatorial function $\mathcal{F}(X)$, a joint spectrum of two or more functions may be computed

Example Let us consider the two functions $\mathcal{F}_1(X)$ and $\mathcal{F}_2(X)$ tabulated below Encoding and combining the two individual outputs we have,

x_3	x_2	x_1	$\mathcal{F}_1(X)$	$F_2(X)$	Combined $\mathcal{F}_1(X)$ and $\mathcal{F}_2(X)$
0	0	0	0	0	0
0	0	1	1	0	2
0	1	0	1	0	2
0	1	1	0	1	1
1	0	0	1	0	2
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	3

The Hadamard transform now gives

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 2 \\ 1 \\ 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 12 \\ -2 \\ -2 \\ 0 \\ -2 \\ 0 \\ 0 \\ -6 \end{bmatrix} \tag{2 16}$$

$$\mathbf{T}^n \qquad \mathbf{Z}_{1+2} = \mathbf{R}_{1+2}$$

2.2.3 The Fast Transform Procedures

The step-by-step execution of Hadamard forward or inverse transform involves the $\pm$ summation of a total of $2^n X 2^n$ individual product terms Thus the computation of spectra using matrix operations is impractical for all but extremely small functions Due to the recursive

structure of [T], however, it is possible to use a fst, in-place transformation algorithm [7], [22], as given below

$$t_0(\omega) = \mathcal{F}(\omega) \quad (2\ 17)$$

$$t_0(2^{n-1}+\omega) = \mathcal{F}(2^{n-1}+\omega) \quad (2\ 18)$$

$$t_i(\omega) = t_{i-1}(2\omega) + t_{i-1}(2\omega+1) \quad (2\ 19)$$

$$t_i(2^{n-1}+\omega) = t_{i-1}(2\omega) - t_{i-1}(2\omega+1) \quad (2\ 20)$$

$$\omega = 0,1, \quad ,2^{n-1}-1 \quad (2\ 21)$$

$$i = 1, \quad ,n \quad (2\ 22)$$

On completion the array t contains the transform Instead of involving the total number of individual product terms, the total number of operations is reduced from 2^nX2^n to 2^nXn, which is expressed as Nlog_2N, where N = 2^n

Example This is illustrated below for the function $\mathcal{F}(X)$ of Fig 2 3 for S spectrum

ω	$\mathcal{F}(\omega)$	$t_0(\omega)$	$t_1(\omega)$	$t_2(\omega)$	$t_3(\omega)$	$t_4(\omega)$	Coeff
0	0	0	-1	-3	-5	-9	s_0
1	-1	-1	-2	-2	-4	3	s_1
2	-1	-1	0	0	3	7	s_2
3	-1	-1	-2	-4	0	3	s_{12}
4	1	1	2	1	3	3	s_3
5	-1	-1	-2	2	4	-1	s_{13}
6	-1	-1	-2	0	3	3	s_{23}
7	-1	-1	-2	0	0	-1	s_{123}
8	1	1	1	1	-1	-1	s_4
9	1	1	0	2	4	3	s_{14}
10	-1	-1	2	4	-1	-1	s_{24}
11	-1	-1	0	0	0	3	s_{124}
12	-1	-1	0	1	-1	-5	s_{34}
13	-1	-1	0	2	4	-1	s_{134}
14	-1	-1	0	0	-1	-5	s_{234}
15	-1	-1	0	0	0	-1	s_{1234}

Thus the obtained spectrum S = $t_4(\omega)$ is exactly the same as the one obtained in the previous case

2.2.4 Properties of S and R Spectral Coefficients

The *principal properties* of the coefficients of the S and R spectrum for completely and incompletely specified Boolean functions for the well-known Walsh-type transform matrices are shown below according to [1], [2], [7], [8], [23], [25]

1 The transformation matrix for each ordering of the Walsh functions is *complete* and *orthogonal*, therefore, there is no information lost in the **S** and **R** spectra concerning the minterms of the Boolean function $\mathcal{F}$

2 Only the Hadamard–Walsh matrix describing the Hadamard–Walsh transform has the recursive *Kronecker product structure* Other possible variants of the Walsh transforms, described by the corresponding matrices, are known in the literature as the Walsh–Kaczmarz, Rademacher–Walsh, and Walsh–Paley transforms

3 Only the Rademacher–Walsh matrix is not *symmetric*, all other variants of Walsh matrices are symmetric, so that, disregarding a scaling factor, the same matrix can be used for both the forward and inverse transform operations

4 Each spectral coefficient s_I or r_I is described by its *order*, *subindexes* and *magnitude* The order of the spectral coefficient is equal to the number of subindices, and the subindices are the subscripts of all variables of a standard trivial function corresponding to the coefficient The magnitude of a spectral coefficient is its value. In this chapter, i, j and k denote subindices, and the order is denoted by o

5 When the classical matrix multiplication method is used to generate the spectral coefficients for different Walsh transforms (different T matrices represent different Walsh functions with different orderings), the only difference in the rtesult is the *ordering* of the spectral coefficients The coded vector **Z** or **Y** corresponding to the original truth vector of a Boolean function is the same for all orderings of Walsh functions The values of the coefficients s_I and r_I with the same subindices are the same for every ordering of Walsh transforms

6 Each spectral coefficient (either s_I or r_I) gives a *correlation value* between the Boolean function $\mathcal{F}$ and a *standard trivial function* u_I corresponding to the coefficient The *standard trivial functions* for the spectral coefficients are respectively

- for the *dc* coefficients (direct current coefficients) s_0 or r_0, the universe of the boolean function denoted by u_0
- for the first order coefficients s_I or $r_I (I = i, i \neq 0)$, the variable of the boolean function $\mathcal{F}$ denoted by u_i

- for the second order coefficients s_I or $r_I(I = ij, i \neq 0, j \neq 0, i \neq j)$, the Exclusive-OR function between variables x_i and x_j of the boolean function $\mathcal{F}$ denoted by u_{ij}
- for the third order coefficients s_I or $r_I(I = ijk, i \neq 0, j \neq 0, k \neq 0, i \neq j, i \neq k, j \neq k)$, the Exclusive-OR function between variables x_i, x_j and x_k of the boolean function $\mathcal{F}$ denoted by u_ijk, etc

7 The number of spectral coeficients of zth order is equal to $C_n^z = \begin{pmatrix} n \\ z \end{pmatrix}$, where n is the number of variables of a Boolean function

8 For a completely specified Boolean function the maximal value of any individual spectral coefficient s_I is $+2^n$ and the minimal value is -2^n These values occur when the Boolean function $\mathcal{F}$ is equal to either a standard trivial function u_I (sign $+$) for the maximal value or to its complement (sign $-$) for the minimal one In either case, all the remaining spectral coefficients have zero values because of the orthogonality of the transform matrix T

9 The maximal value of any spectral coefficient r_I except r_0 is $+2^{n-1}$ and will result when the Boolean function $\mathcal{F}$ is equal to complement of a standard trivial function u_I The minimal value is -2^{n-1} and will result when $\mathcal{F}$ is equal to a standard trivial function u_I In either case, all the remaining spectral coefficients have zero values because of orthogonality of the transform matrix T

10 For an incompletely specified Boolean function the maximal value of any individual spectral coefficient s_I is $+2^n - 1$, and the minimal one is $-2^n + 1$ These occur when the Boolean function $\mathcal{F}$ is equal to either a standard trivial function for the maximal value of s_I or to its complement for the minimal value, in all but one minterm which is a don't care

11. The maximal value of r_0 spectral coefficient is $+2^n$ and will result when the Boolean function $\mathcal{F}$ is a *tautology* The minimal value is -2^n and will result when $\mathcal{F}$ is equal to the complement of the *tautology* The *tautology* is the logical function for which all the minterms are true

12 When, for more than half of the spectraol coefficients of any completely specified Boolean function $\mathcal{F}$, the majority of the minterms have the same logical values as the

minterms of standard trivial functions, the sum of all the coefficients of the S spectrum has a maximum value equal to $+2^n$ When, for more than half of the spectraol coefficients, the majority of the minterms of $\mathcal{F}$ have the complemented logical values to the minterms of standard trivial functions, the sum of all the coefficients of the S spectrum has a minimal value equal to -2^n

13. For any incompletely specified Boolean function, the sum of all coefficients of the S spectrum has a maximal value of $+2^n - 1$ and a minimal value of $-2^n + 1$ The maximal or minimal value happens when the Boolean function has exactly one don't care minterm and all the spectral coefficients follow the rule of property 12

14. With the exception of u_0, every standard trivial function u_I corresponding to an n variable Boolean function $\mathcal{F}$ has the same number of true and false minterms That number is equal to 2^{n-1}

15. For each true minterm the coefficients from the spectrum S are $s_0 = 2^n - 2$, and all remaining $2^n - 1$ spectral coefficients s_I are equal to ± 2 The method for choosing the signs of spectral coefficients is described in Section 2 4

16. The spectrum S of each false minterm is given by $s_I = 0$

17. For each don't care minterm the coefficients from the spectrum S are $s_0 = 2^{n-1} - 1$, and all remaining $2^n - 1$ spectral coefficients s_I are equal to ± 1 The method for choosing the signs of spectral coefficients is described in Section 2 4

The new method of calculating spectral coefficients of Boolean functions described in the Section 2 4 needs the representation of Boolean functions in the form of arrays of disjoint cubes The method gives the correct values of spectral coefficients independently from the shape and size of disjoint cubes in the array of cubes as long as all the minterms of the function are covered only once by a cube The next section describes the algorithm that generates this kind of representation of Boolean functions originally represented either as a list of true/false minterms or as an already minimized SOPE But before we describe the algorithm, here are some notations and definitions [26]

Definition 1 A *cube* is a set C of literals such that $x \in C$ implies that $\vec{x}$ is not an element of C (*e.g* $\{a, b, \vec{c}\}$ is a *cube* but $\{a, \vec{a}\}$ is not a *cube*) A *cube* represents the

conjuction of its literals. In a *cube*, literals 0 and 1 are referred to as *bound* coordinates and x (don't care) as *free*

Definition 2 A *ON-cube* is a cube such that the Boolean function which consists of this *cube* evaluates to 1 for the combination of the input variables represented by this cube Similarly we have *OFF-cubes* and *DC-cubes*

Definition 3 A *vertex* is a special cube in which all coordinates (literals) are 0 or 1

Definition 4 A cube is C_a is said to *contain* another cube C_b, if C_b can be obtained from C_a by appropriately changing all don't cares (represented here by x) in the cube C_a to 1s or 0s ($e\ g$ $10x$ contains both the cubes 101 and 100) Note that, both the cubes C_a and C_b should be of the same type ($i\ e$ *ON*, *OFF* or *DC*)

Definition 5 The *intersection* of cubes C_a and C_b is defined as the smallest cube containing all of their common vertices – it is contained in all cubes containing them If they have no common vertices, they are said to be *disjoint* Using the symbol I for intersection, $(1x1)I(0x1) = \phi$ Thus the cubes $1x1$ and $01x$ are disjoint Similarly, $(x10x)I(01xx) = 010x$

Definition 6 The *sharp product* (termed as #-product) of cube C_a with C_b is a differencing product, consisting of a set of vertices of C_a which are not contained in C_b Again we assume that both the cubes are of the same type There are three rules

- If C_a and C_b are disjoint, then $C_a \# C_b = C_a$,
- If C_a is contained in C_b, then $C_a \# C_b$ contains no cubes, $i\ e$ $C_a \# C_b = \phi$
- If neither of the above conditions hold, then the output will be a set of certain faces of C_a that are not in C_b For each coordinate i for which $C_a(i) = x$ and $C_b(i) = 1$ or 0, there will be a cube C_d in the #-product having $C_d(i) = \bar{e}$ and all other coordinates identical to those of C_a

  Example

$$(1x1x)\#(x1x0) = 101x,\ 1x11,$$
$$(xxx)\#(101) = 0xx,\ x1x,\ xx0$$

Definition 7 The *disjoint sharp* operation (termed as $\#_j$-product) aids in the counting the number of vertices When $\#_j$ is formed between two cubes C_a and C_b (of same type) it

obtains a set of disjoint cubes. To effect $C_a \#_j C_b$, first the ordinary $C_a \# C_b$ is formed Let the first cube C_1 produced be as with the #-product, the second cube C_2 is modified so that in the first coordinate $i(1)$ where C_1 has a 1 (or 0) and C_2 has an x, this x is changed to an opposite 0 (or 1) in the $\#_j$ product For the third cube C_3 produced by the #-product, the $i(1)$ coordinate is made similarly equal to 0 (or 1), likewise the second coordinate $i(2)$ is found where C_2 has a 1 (or 0) and C_3 has an x This x is changed to a 0 (or 1), rendering it disjoint from C_1 and C_2

Example

	$xxxx$		$xxxx$
#	1011	$\#_j$	1011
	$0xxx$		$0xxx$
	$x1xx$		$11xx$
	$xx0x$		$100x$
	$xxx0$		1010

2.3 Algorithm to Generate Disjoint Cubes

As will be seen the calculation time for Hadamard spectrum increases linearly with the number of disjoint cubes Thus, the determination of a quasi-minimal number of cubes in the disjoint cube representation of a Boolean function is crucial for the effective calculation of its spectrum

Several algorithms for the generation of arrays of disjoint ON- and DC–cubes (if any) or an array of disjoint OFF–cubes were described and used in ESPRESSO [20], [24], EXORCISM [27] and PALMINI[28] Here the algorithm and its implementation as described in [24] are improved The drawbacks of this algorithm are also overcome.

A logical function is represented in the arrays of cubes form It is enough to have only any two of the three possible sets of cubes (ON, OFF and DC) In the following presentation of the algorithm it will be assumed that only the ON- and DC–arrays are available (the same as assumed in [20]) This algorithm can be applied to the ON-array and DC-array separately maintained as unidirectional linked lists

The random ordering of cubes is replaced by a list of cubes in which the are sorted according to their size Every time two cubes C_a and C_b are compared The algorithm uses two pointers a and b which correspond to the actual position of cubes C_a and C_b in the list The pointer a indicates the position of the cube which is the first of the pair of the cubes being compared, the pointer b indicates the position of the second cube in the pair The

pointer a changes its values till the $(n-1)th$ node (n = no of nodes in the list) is reached and b changes its values from $2nd$ to nth node accordingly At a given moment, the value of the pointer b is always greater by at least one than the value of the pointer a During the execution of the algorithm the number of nodes in the list can be different from the original number of cubes due to the following reasons

- the disjoint sharp operation ($\#_j$-operation) can generate more than one cube as its result
- the contain operation can remove some cubes

As a last step of the algorithm, the cubes are merged, where possible, to obtain a smaller total number of disjoint cubes

Algorithm Generation of a set of disjoint cubes from the set of non-disjoint cubes

```
initialise
a = head of the list,
if(a == end-of-list) stop, /* if the list is empty */
b = a → next,
if(b == end-of-list) stop, /* if the list has only one node */
do {
    d = step = 1,
    count = 0,
    do {
        if C_a and C_b are not disjoint
            {
            if C_a contains C_b
                {
                remove C_b from the list and
                update the list,
                }
            else
                {
                do C_b #_j C_bj
                d = no of cubes resulted from C_b #_j C_bj
                if(count == 0) step = d,
                replace C_b by the first solution cube,
                Insert the other solution cubes in the list to the next of C_b,
                b = b + d - 1, /* advance b so that it points to
                                  the node containing the last solution cube */
                }
            }
        b = b → next,
        count + +,
    } while (b ≠ NULL), /* until the last node is processed */
```

$b = a + step + 1$,
$a = a \rightarrow$ next,
} while ($a \neq$ end-of-list),
merge the cubes,

Example The above algorithm is applied to the function of Fig 2.3. The steps of the execution are shown in Fig 2.4. Fig 2.4(a) shows the minimized function of Fig 2.3 minimized by ESPRESSO [20]. A '*' in the list of cubes indicates that the disjoint sharp operation ($\#_j$) has to be perfomed between those two cubes. Fig 2.4(e) is the list after the disjoint operation. Fig 2.4(f) is the list after merging the possible cubes.

*$xx1x$	ON		*$xx1x$	ON		*$xx1x$	ON	
$11xx$	ON		$110x$	ON		$110x$	ON	
$x1x1$	ON		*$x1x1$	ON		$x101$	ON	
$00x1$	ON		$00x1$	ON		*$00x1$	ON	
0000	DC		0000	DC		0000	DC	
(a)			(b)			(c)		
$xx1x$	ON		$xx1x$	ON		$xx1x$	ON	
$110x$	ON		$110x$	ON		$110x$	ON	
$x101$	ON		0101	ON		$0x01$	ON	
0001	ON		0001	ON		0000	DC	
0000	DC		0000	DC				
(d)			(e)			(f)		

Figure 2.4 Stages of execution of the algorithm to generate a disjoint cube representation

2.4 New Method for Calculation of Spectrum

An algorithm already exists for calculating spectral coefficients for completely specified Boolean functions directly from a SOPE [2], [29]. When the implicants are not mutually disjoint, this algorithm requires an additional correction to calculate the exact values of spectral coefficients for minterms of Boolean function $\mathcal{F}$ that are included more than once in some implicants. But in the disjoint cube representation, the exact values of spectral coefficients can be calculated immediately without having to perform correction operations. Here the extension of the algorithm to incompletely specified Boolean function is proposed.

Definition 1 The *cube of degree m* is a cube that has m literals that can be either in affirmation or negation (*i.e.*, m is equal to the sum of the number of zeroes and ones in

the description of a cube) Let p denote the number of x's in the cube and let n denote the number of variables of a Boolean function Then, $n = m + p$

Example The cube $1x00$ is of degree 3 since three of the literals describing this cube are either in affirmation (x_1) or negation (x_3 and x_4) The cube does not depend on literal x_2

Definition 2 The *partial spectral coefficient* of an ON- or DC–cube with degree m of a Boolean function $\mathcal{F}$ is equal to the value of the spectral coefficient that corresponds to the contribution of this cube to the full n-space spectrum of $\mathcal{F}$ The number of partial spectral coefficients *npsc* describing the $\mathcal{F}$ is equal to the number of ON- and DC–cubes describing this function

Example Considering Tables 2 1 and 2 2 representing this method of calculation for the same function of Fig 2 4, each row shows the partial spectral coefficients of either On- or DC–cubes of a Boolean function This function has four partial spectra, which is equal to the number of disjoint ON- and DC–cubes describing this function ($npsc = 4$)

$x_1x_2x_3x_4$		s_0	s_1	s_2	s_3	s_4	s_{12}	s_{13}	s_{14}	s_{23}	s_{24}	s_{34}
$x1xx$	ON	0	0	16	0	0	0	0	0	0	0	0
$x011$	ON	12	0	-4	4	4	0	0	0	4	4	-4
$10x0$	ON	12	4	-4	0	-4	4	0	4	0	-4	0
0000	DC	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
		-9	3	7	3	-1	3	-1	3	3	-1	-5

Table 2 1. Table showing DC, *1st* and *2nd* order coefficients

$x_1x_2x_3x_4$		s_{123}	s_{124}	s_{134}	s_{234}	s_{1234}
$x1xx$	ON	0	0	0	0	0
$x011$	ON	0	0	0	-4	0
$10x0$	ON	0	4	0	0	0
0000	DC	-1	-1	-1	-1	-1
		-1	3	-1	-5	-1

Table 2 2 Table showing *3rd* order coefficients

Suppose the lists of disjoint ON- and DC–cubes that fully define $\mathcal{F}$ are given Then each cube of degree m can be treated as a minterm within its particular reduced m-space of $\mathcal{F}$ The spectrum of each true minterm is given by $s_0 = 2^n - 2$, and all remaining

$2^n - 1$ coefficients are equal to ± 2 (property 15) Similarly, the spectrum of each don't care minterm is given by $s_{DC0} = 2^{n-1} - 1$, and all the remaining $2^n - 1$ coefficients are equal to $\pm/, 1$ (property 17) The symbols s_{DCI} denote spectral coefficients corresponding to DC–cubes (when $I = 0$, the symbol s_{DC0} denotes a DC spectral coefficient corresponding to a DC–cube)

Cubes of degree m have the following properties

1 The contribution of an ON–cube of degree m to the full n-space spectrum of function $\mathcal{F}$(where n is the number of variables in the function $calf$) is represented as follows

$$s_0 \, in \, full \, n-space = 2^n - 2 \times 2^p \tag{2 23}$$

and

$$s_I \, in \, full \, n-space = s_I \, in \, m - space \times 2^p \tag{2 24}$$

where $I \neq 0$

2 The contribution of an DC–cube of degree m to the full n-space spectrum of function $\mathcal{F}$ is related as follows

$$s_{DC0} \, in \, full \, n-space = 2^{n-1} - 2 \times 2^p \tag{2 25}$$

and

$$s_{DCI} \, in \, full \, n-space = s_{DCI} \, in \, m - space \times 2^p \tag{2 26}$$

where $I \neq 0$ When the above formulas are applied to minterms ($i \, e$ for $m = n$ and $p = 0$) they reduce to properties 15 and 17. The contribution of a DC–cube of degree m is equal to one half of the contribution of an ON–cube that has the same degree m

Equations 2 24 and 2 26 determine the absolute values of those partial spectral coefficients s_I that are not equal to zero for a given cube Properties 3-4 determine the signs of the partial spectral coefficients, and whether some of them are equal to zero

Example Considering Tables 2 1 and 2 2 again, the value of partial spectral coefficient s_0, corresponding to the On–cube $10x0$ ($n = 4, p = 1$) is equal to $2^4 - 2 \times 2^1 = 12$ according to Equation 2 23 The absolute values of those partial spectral coefficients s_I that are not equal to zero are calculated according to Equation 2 24 and are equal to $2 \times 2^1 = 4$

3 If in a given cube the x_i variable of a Boolean function is denoted by the symbol "x", then all of the partial spectral coefficients s_I whose indices I contain the subindex i are equal to zero

4 For an ON- or DC–cube of degree m the number of nonzero partial spectral coefficients is equal to 2^{n-p}, except for $p = n - 1$ when there is only one nonzero partial spectral coefficient

Example Consider Tables 2 1 and 2 2 again The variable x_3 is denoted by symbol x in the cube $10x0$ So all partial spectral coefficients with subindex 3 are equal to 0 Therefore, $s_3 = s_{13} = s_{23} = s_{34} = s_{123} = s_{134} = s_{234} = s_{1234} = 0$ For this cube by property 4, the number of non-zero partial spectral coefficients is equal to $2^{4-1} = 8$

The cube $x1xx$ has three variables denoted by x x_1, x_3 and x_4 Therefore, by property 3 and 4, only the partial spectral coefficient s_2 is non-zero

The following properties describe the signs of each partial spectral coefficient s_I, where $I \neq 0$

5 If in a given cube the x_i variable of a Boolean function is an affirmation, then the sign of the corresponding first-ordert coefficient is positive, otherwise for a variable that is in negation, the sign of the corresponding first order coefficient is negative

6 The signs of all even-order coefficients are given by multiplying the signs of the related first-order coefficients by -1

7 The signs of all odd-order coefficients are given by multiplying the signs of the related first-order coefficients

Example Considering the Tables 2 1 and 2 2 again, in the ON–cube $10x0$ the variable x_1 is in affirmation, while the variables x_2 and x_4 are in negation Therefore, the sign of the partial spectral coefficient s_1 is positive and the signs of partial spectral coefficients s_2 and s_4 are negative

The sign of the even-order(2) partial spectral coefficient s_{12} of cube $10x0$ is positive, since the sign is determined by the related first-order coefficients, s_1 and s_2, times -1, *i e* $(-1) \times 1 \times (-1) = 1$

The sign of the odd-order(3) partial spectral coefficient s_{124} of the same cube $10x0$ is positive, since it is determined by the related first-order coefficients, s_1 ,s_2 and s_4 i e $1 \times (-1) \times (-1) = 1$

The algorithm is as follows

Algorithm Calculation of spectral coefficients for completely and incompletely specified Boolean functions

1 For each ON- and DC–cube of degree m, calculate the value and sign of the contribution of this cube dto the full n-space spectrum according to the properties described previously

2 The values of all spectral coefficients s_I, except s_0 are equal to the sum of all the contributions to the spectral coefficients from all ON- and DC- disjoint cubes

3 For a completely specified Boolean function the value of the dc spectral coefficient s_0 is equal to the sum of all the partial spectral coefficients corresponding to all of the disjoint ON–cubes describing the function, plus the correction factor $-(k-1) \times 2^n$, where k is the number of disjoint cubes in the list of ON–cubes

4 For an incompletely specified Boolean function the value of the dc spectral coefficient s_0 is equal to the sum of all the partial spectral coefficients corresponding to all of the disjoint ON- and DC–cubes describing the function, plus the correction factor $-(k-1) \times 2^n - w \times 2^{n-1}$, where k is the number of disjoint ON-cubes and w is the number of disjoint DC–cubes

The correction factor $-(k-1) \times 2^n$ compensates for the fact that the cubes over the complete n-space have been added k times during the calculation of the k partial spectral coefficients A similar explanation applies to DC–cubes as well

The main advantage of this algorithm is that it can calculate each coefficient separately or in parallel So if some of the 2^n spectral coefficients are not needed for a particular application, then a reduced number of operations can be performed

Example An example of the calculation of the S spectrum for the four-variable incompletely specified Boolean function is shown in Tables 2 1 and 2 2 Fig 2 4(f) shows the input data for the algorithm of this section The values and signs of all the

partial spectral coefficients for this function has been determined by the properties 1-7 The results are shown for two cubes

The spectral coefficients of the first ON–cube in Tables 2 1 and 2 2 (cube $10x0$ of degree $m = 3$) are as follows

1 within its own m-space, treated as a single minterm;

$$s_0 = 6, s_1 = 2, s_2 = -2, s_3 = 0$$

$$s_4 = -2, s_{12} = 2, s_{13} = 0, s_{14} = 2$$

$$s_{23} = 0, s_{24} = -2, s_{34} = 0, s_{123} = 0$$

$$s_{124} = 2, s_{134} = 0, s_{234} = 0, s_{1234} = 0$$

2 within the full n-space of Boolean function $\mathcal{F}$(partial spectral coefficients),

$$s_0 = 12, s_1 = 4, s_2 = -4, s_4 = -4$$

$$s_{12} = 4, s_{14} = 4, s_{24} = -4, s_{124} = 4$$

All other coefficients for this cube are zero On the other hand, the spectral coefficients of the DC–cube (cube 0000 of degree $m = 4$, *i e* , single minterm) are as follows

1 within its own m-space, treated as a single minterm,

$$s_0 = 7, s_1 = -1, s_2 = -1, s_3 = -1$$

$$s_4 = -1, s_{12} = -1, s_{13} = -1, s_{14} = -1$$

$$s_{23} = -1, s_{24} = -1, s_{34} = -1, s_{123} = -1$$

$$s_{124} = -1, s_{134} = -1, s_{234} = -1, s_{1234} = -1$$

2 within the full n-space of $\mathcal{F}$(partial spectral coefficients) the same as within its own m-space since it is a single minterm

In order to obtain the values of all of the spectral coefficients of the whole function, except s_0, the columns of partial spectral coefficients corresponding to all cubes describing the function are added (step 2 of the algorithm) The value of s_0 is obtained by the addition of all partial spectral coefficients with the correction factor(step 4) Step 3 is not performed because it is an incompletely specified function

The resulting spectrum is shown in the bottom row of Tables 2 1 and 2 2 and, as can be easily checked, is exactly the same as obtained by matrix multiplication

2.4.1 Spectral Coefficients for Systems of Boolean Functions

The algorithms from the previous sections can be modified easily to calculate Walsh spectra of systems of Boolean functions In this section, we describe a method to calculate spectra of a system of incompletely specified Boolean functions

Let us assume that the functions in the system are in the order $\mathcal{F}[1]$, $\mathcal{F}[2]$, $\mathcal{F}[3]$, , $\mathcal{F}[b]$, where b is the number of functions in the system and the function $\mathcal{F}[b]$ is on the rightmost position of the system Then, for a system of incompletely specified functions, the following properties hold

1 The contribution of the spectrum of the function $\mathcal{F}[i]$, $i = 1, 2,$ „b to the total spectrum of a system of Boolean functions S_{TOT} is equal to the spectrum $S_{I[i]}$ of the function $\mathcal{F}[i]$ calculated by the algorithm of the previous section, which in turn has to be modified by Equation 2 27

2 The total spectrum of a system of Boolean functions S_{TOT} is equal to the sum of all the modified spectra of all the Boolean functions in the system

The contribution of the spectrum of the ith function $\mathcal{F}[i]$ to the total spectrum of a system of b Boolean functions is denoted in Equation 2 27 by $S^*_{I[i]}$, and the spectrum of the ith function calculated by the algorithm of the previous section is denoted by $S_{I[i]}$

$$S^*_{I[i]} = 2^{b-i} \times S_{I[i]} \tag{2 27}$$

Conversion of **S***-spectra to* **R***-spectra for a system of Boolean functions* The individual first and higher order coefficient values in **S** and **R** for a system of Boolean functions are related linearly as in Equation 2 13 However, for the DC-coefficient(s_0), the relationship is given by,

$$r_0 = \frac{1}{2}\sum_{i=1}^{b} 2^{b-i} s_{0[i]} \tag{2 28}$$

Example As an example we consider the two functions of section 2 2 2 (page 11) The steps of the calculations are shown in Table 2 3 First, the S-spectra of the individual functions are calculated using the algorithm of the previous section and are shown in the first two rows of the table The modified value of the spectrum of function $\mathcal{F}[1]$ is shown in the third row of the table For the function $\mathcal{F}[2]$ the modified value of spectrum is equal to the original one Then using Equation 2 27 the combined S-spectra is calculated and is

shown in the fourth row This combined S-spectra is then converted to R-spectra, which is shown in the last row of the table

	s_0	s_1	s_2	s_3	s_{12}	s_{13}	s_{23}	s_{123}
$S_{[2]}$	0	4	4	4	0	0	0	-4
$S_{[1]}$	0	0	0	0	0	0	0	8
$S^*_{[1]}$	0	0	0	0	0	0	0	16
S_{TOT}	0	4	4	4	0	0	0	12
R_{TOT}	12	-2	-2	-2	0	0	0	-6

Table 2 3 Table showing calculation steps of spectral coefficients the two functions

Chapter 3

PLA DECOMPOSITION

Programmable Logic Arrays(PLAs) are important subsystems in the design of digital integrated circuits A PLA provides a simple and regular layout strategy for Boolean equations expressed in two level canonical form, and is usually used to implement *random* logic (random in the sense that the designer sees no regular structure in the Boolean equations) Typical examples are the controllogic for a *RISC*, or the control logic for the microsequencer of a microcoded machine With the addition of latches for feedback, PLAs are often used for the combinatorial logic in a finite state machine The optimization of PLAs is a useful application of computer aided design to the automated synthesis of custom VLSI designs

Techniques for optimizing the structure of a PLA have become well understood The optimization goals are to minimize the area occupied by the PLA and to minimize the delay through the PLA The regular structure of a PLA means that the area of the PLA is simply proportional to the number of product terms in the array, and to a first order approximation, the delay through the PLA is also proportional to the number of product terms (*i e* , independent of the structure of each product term)

A PLA macrocell can use *input decoders* which group the input signals into pairs [30], [31] When the inverters of a standard PLA is replaced with decoders, it is called the *PLA with decoders* The standard PLA is a special case of a PLA with decoders, being a PLA with one-bit decoder The four decodes for each pair are used in the core of the PLA rather than the signals and their complements An example of a PLA with input decoders is shown in Fig 3 1 A PLA which uses input decoders can always be built with no more rows than required by the normal PLA structure [32] However, the number of literal lines (columns) is equal to that of a standard PLA So, the array size of a PLA with two bit decoders never exceeds that of a standard PLA Table 3 1 shows the number of rows needed to realise

various classes of functions [32]

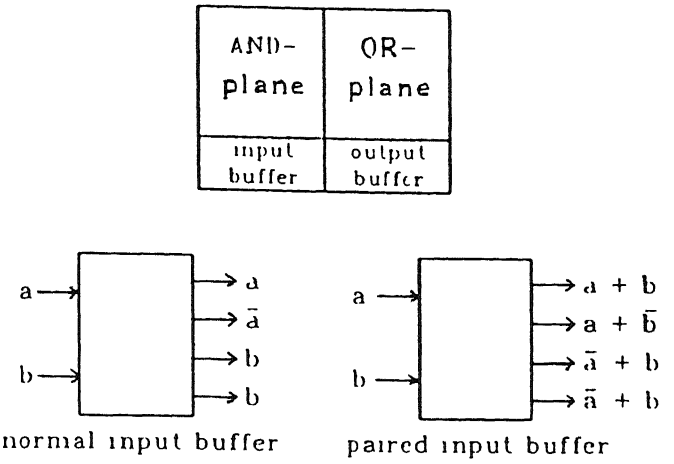


Figure 3 1 PLA using input decoders

	Standard PLA	PLA with two-bit decoders
Arbitrary function (worst case)	2^{n-1}	2^{n-2}
Symmetric function (worst case)	2^{n-1}	$3^{(n-2)/2}$
Parity function (worst case)	2^{n-1}	$2^{(n-2)/2}$
Random function of 10-variables(average)	163	120

Table 3 1 Number of columns to realize n-variable functions by PLAs (n is even)

In this chapter we will discuss the linear decomposition of PLAs which involves separation of the linear (that can be implemented with EX-OR gates alone) and nonlinear (which requires a complete basis such as AND, OR, NOT gates, or NAND gates) blocks [7], [33] It leads to a cascaded implementation with a linear preprocessor This implementation has reduced complexity and is accompanied by an improved testability since the linear part is completely testable for single-stuck-at faults and the nonlinear part, implemented as a

standard PLA has reduced complexity

3.1 Notations and Definitions

Let us consider an m-input, k-output Boolean function $f \quad \{0,1\}^m \rightarrow \{0,1\}^k$ The k outputs are denoted as $f_{k-1}, f_{k-2}, \quad , f_0$ so that using decimal indices, $x^{\cdot} = (x_{m-1}, \quad , x - 0) \in \{0,1\}^m$ can be expressed as $x = \sum_{i=0}^{m-1} x_i 2^i$ and $f = (f_{k-1}, \quad , f_0) \in \{0,1\}^k$ by $f = \sum_{i=0}^{k-1} f_i 2^i$ The symbols x and f will be used interchangeably to denote the binary vectors as well as their decimal representations, for convenience of notation

The output of f can be described as a vector as in chapter 2, with entries specifying the outputs for each input vector $x, 0 \leq x \leq 2^m - 1$

$$F = [f(0), f(1), \quad , f(2^m - 1)]^T, \tag{3 1}$$

where $f(x)$ takes on integer values, and

$$F_i = [f_i(0), f_i(1), \quad , f_i(2^m - 1)]^T, \tag{3 2}$$

where $f_i(x), 0 \leq i \leq k-1$, takes on binary values, and the superscript T indicates transpose of the vector

Definition The *autocorrelation* of f is defined for its individual outputs f_i is given by,

$$B^{(f_i,f_i)}(\tau) = \sum_{x=0}^{2^m-1} f_i(x) f_i(x \oplus \tau) \tag{3 3}$$

$$B^{(f,f)}(\tau) = \sum_{i=0}^{k-1} B^{(f_i,f_i)}(\tau), 0 \leq \tau \leq 2^m - 1 \tag{3 4}$$

$B^{(f_i,f_i)}(\tau)$ can also be computed by the application of the *Weiner-Khinchin* theorem [7] applying the Walsh-Hadamard transform twice

$$B^{(f_i,f_i)}(\tau) = 2^{-m} W(W(f_i))(\tau), 0 \leq tau \leq 2^m - 1 \tag{3 5}$$

Where, $W(f)$ is the Walsh-Hadamard transform of f

The autocorrelation is a measure of the similarity of function outputs at a certain distance apart in the Boolean space For example, $\sum B^{(f,f)}(\tau)$ for all τ with Hamming weight 1, is a measure of the number of true minterms at Hamming distance 1, and hence can be considered as the closeness of 1's in a Karnaugh map of the function (Hamming

weight of a *vertex* is the number of 1's in the vertex and Hamming distance between two vertices is the number of bits in which they differ from each other)

The definition of autocorrelation can be used to formulate complexity of a function by substituting $\tau = 1, 2, 4, \quad , 2^{m-1}$ (of Hamming distance 1), because these are the number of true minterms at distance 1 Thus,

$$\psi(f) = \sum_{\tau=1,2,4,\quad,2^{m-1}} B^{f,f}(\tau) \tag{3 6}$$

is a measure of the function's simplicity The reason for using this criterion is that two minterms that are at unit distance can be combined to form an *implicant*, and hence, amounts to a reduction in the literal count

3.2 Linear Decomposition

The optimal linear decomposition problem is stated as follows

Find a logic block σ, consisting of EX-OR gates, that will minimize the complexity measure of the block f_σ (Fig 3 2) to be implemented over a complete basis

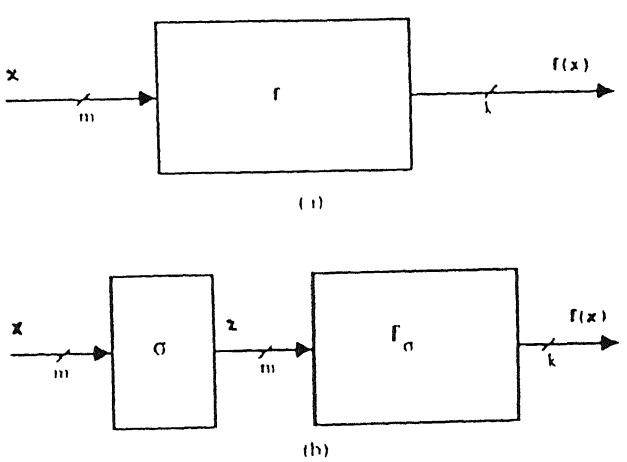


Figure 3 2 (a) Original, (b) Decomposed implementation of a function

The effect of the EX-OR block is a simple basis translation, a rotation of the Boolean space in which f is defined, so as to maximize the criterion ψ in the new orientation The preprocessor σ and the core function f_σ can be constructed by the following procedure [5]

Procedure Let T be an $m \times m$ matrix with columns τ_i so that $T = [\tau_0, \tau_1, \ldots, \tau_{m-1}]$ (τ_i is $m \times 1$), where τ is found as follows

1 $B^{(f,f)}(\tau_0) = max(B^{(f,f)}(\tau)), \tau \neq 0$

2 Let $L_0 = \{0, \tau_0\}$

τ_1 is then obtained as follows

$$B^{(f,f)}(\tau_1) = max(B^{(f,f)}(\tau)), \tau \in L_0$$

3 Once τ_0, τ_1 have been found, τ_s can be determined as,

$$B^{(f,f)}\tau_s = max(B^{(f,f)}(\tau)), \tau \in L_{s-1}$$

where,

$$L_{s-1} = \{L_{s-2}, \sum_{i=0}^{2^{s-1}} (\{L_{s-2}\} \oplus \tau_{s-1})\}$$

Here, $\oplus$ implies bitwise EX-OR of τ_{s-1} with each element of L_{s-2}

4 $\sigma = T^{-1}$, where the matrix inversion is carried out mod-2 over the binary numbers $\{0, 1\}$

Once the matrix σ is determined, the linearpart can easily be implemented to convert any given vector x to z, by inspecting the rows of σ A direct implementation of the linear part is given by $z_{m-i-1} = \oplus_{j=0}^{m-1} \sigma_{ij} x_{m-j-1}, 0 \leq i \leq m$ Obviously, each EX-OR gate implementing a z_i requires as many inputs as there are 1's in the ith row of σ In the worst case, the matrix σ will contain $m(m-1)$ nonzero entries Hence, the complexity of implementing the linear part grows proportional to the square of the number of variables in the worst case

The nonlinear part can be computed as follows Multiply every minterm $(x_{m-1}, \ldots, x_0)$ by the matrix σ This new vector is a minterm for the nonlinear part f_σ.

It is easy to see how this procedure is able to reduce the logic complexity of a function The logic simplicity ψ was defined earlier on the basis of autocorrelation If the maximal autocorrelation coefficients correspond to coordinates with Hamming weight 1, then the function has relatively low logic complexity because a large number of the minterms are at Hamming distance 1 If this is not the case then shifting these coefficients to coordinates with unit Hamming distance will produce a new function with low complexity which can

be generated from the original function by a basis translation The new function (with reduced complexity) is f_σ and the translation is done by the matrix σ Whether such a translation will result in substantial improvements in logic complexity depends on the nature of the function itself If all the autocorrelation coefficients are of the same order of magnitude than the function outputs do not exhibit any significant correlation Randomly generated functions and many types of control functions belong to this category On the other hand, arithmetic functions, error correcting logic, and many signal processing functions have highly correlated outputs and seem to benefit from such a decomposition

Example Let us consider a two-bit adder $f(x_3, x_2, x_1, x_0) = 2(x_3 + x_1) + x_2 + x_0$ According to Equation 3 1, $F = [0,1,2,3,1,2,3,4,2,3,4,5,3,4,5,6]^T$, where the ith row of F is the decimal notation og the 3-bit binary output for the adder for the ith row of the truth table

The autocorrelation is computed by Equation 3 4,

$$B^{(f,f)} = [22,8,10,6,8,16,6,14,10,6,18,4,6,14,4,12]^T$$

Using the procedure, after discarding $B^{(f,f)}(0)$, the maximum over the rest of the rows is 18, corresponding to row number $\tau_0 = 10$ (decimal) = 1010 (binary). $L_0 = \{0,10\}$ Without considering the rows of $B^{(f,f)}$ that correspond to members of L_0, the next highest correlation coefficient is 16, in row 5 Hence, $\tau_1 = 5 = 0101, L_1 = \{0,10,5,15\}$

Similarly, $\tau_2 = 7$ or 13 We can choose any one Choosing 13 (1101) arbitrarily, $L_2 = \{0,10,5,15,13,7,8,2\}$ L_2 remains same whether we choose 13 or 7 because the set L_2 contains the linear combinations of both 13 and 7 Similarly, $\tau_3 = 1 = 0001$ Then,

$$T = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \tau_0 & \tau_1 & \tau_2 & \tau_3 \end{bmatrix}$$

$$\sigma = T^{-1} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

The block implementing σ (Fig 3 3) translates $x \in \{0,1\}^4$ to $z \in \{0,1\}^4$ The truth table for f can be translated with respect to σ as follows

$$f_\sigma(z) = f(x) = f_\sigma(\sigma x),$$

$$f_\sigma(x) = f(Tz)$$

For example, let $x = (1101) = 13$

$$Tz = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

so that $f(1101) = f_\sigma(0010) = f_\sigma(2) = 4(= 100)$

$F_\sigma = [0,1,4,3,2,1,2,3,4,5,4,3,6,5,2,3]^T$ The function f_σ is then minimized using

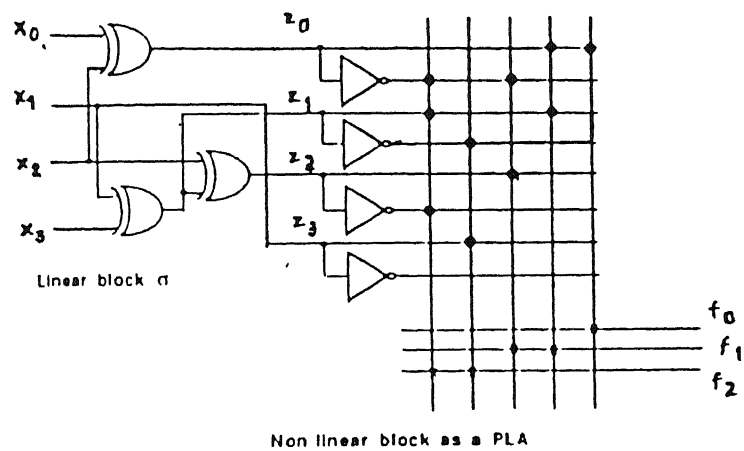


Figure 3 3 Linearized implementation of a 2-bit adder

ESPRESSO [20] and the cover is given below

z_3	z_2	z_1	z_0	f_2	f_1	f_0
-	0	1	0	1	0	0
1	-	0	-	1	0	0
-	1	-	0	0	1	0
-	-	1	1	0	1	0
-	-	-	1	0	0	1

The implementation of f as a serial combination of σ and f_σ is given in Fig 3.3 The number of literals in the two level SOPE of f is 43 and for f_σ it is 15 The number of product terms for f and f_σ are 11 and 5, respectively

The above procedure finds the linear decomposition for an exhaustive list of the outputs of the function However, for an incompletely specified Boolean function the autocorrelation can be found out by using Equation 3 5 and the calculation of Walsh-Hadamard transform can be done by using the algorithms developed in the previous chapter The overhead for the serial implementation is the number of EX-OR gates used to implement σ, which may also be implemented by PLAs However, the reduction in literal count, however, is more direct and results in an improved multilevel implementation [34]

3.3 Results

The procedure described in the previous section has been implemented as a computer program "LINDEC", written in 'C' The effect of serial linearization on some commonly used functions is illustrated in Table 3 2 All the PLAs in the table were minimized using ESPRESSO In case of the decoded PLA the pairs were chosen by the ESPRESSO "-do pair" option It was found that functions that belong to the class of error correcting and translating logic such as code converters, adders and functions with imbedded additions benefit most from this approach Symmetric functions which are usually difficult to implement efficiently by PLAs also benefit from linearization Randomly defined functions are not considerably affected by linear decomposition

Function	No of		No of rows		Decoded PLA		After linear decomposition	
	Inputs	Outputs	Before minimization	After minimization	No of rows	No of decoders	No of rows	No of 2-bit EX-ORs
Full adder	3	2	8	7	4	1	4	2
2-bit adder	4	3	16	11	5	2	5	3
3-bit adder	6	4	64	31	10	3	9	6
4-bit adder	8	5	256	75	17	4	15	7
3-bit multiplier	6	6	64	33	24	3	33	0
4-bit multiplier	8	8	256	127	89	4	123	1
Binary to BCD code converter 0 99	7	8	100	45	33	3	33	6
BCD to binary code converter 0 99	8	7	100	36	30	3	36	0

Table 3 2 Effect of serial decomposition on some functions

Chapter 4

FAULT DETECTION IN PLAs

The design of circuits based on PLAs as multiple-output sum-of-products expressions has become very popular [35], since the geometric layout properties of PLAs are useful in VLSI designs However, the testing of VLSI circuits in general poses difficult problems due to the higher density of the circuits and consequent increase in testing complexity For PLAs in particular, the methods of random pattern testing have not been proved effective and the complete task of test set generation still remains the main option

In classical testing a list of fault-free response of the PLA to the test set is required For most practical cases, the large storage requirement of such a list makes such test procedures very expensive Moreover, the computational cost to generate the test set increases exponentially with the circuit size Syndrome testing [36] is a very simple yet effective coding which counts the number of logical 1's appearing at the output of a circuit The storage problem is solved in this method because only the syndrome of the fault-free circuit has to be stored Also, the expensive stage of test generation and fault simulation is eliminated When a fault causes the syndrome to change, the circuit is syndrome testable for that fault it is said that the test covers the fault Two-level irredundant circuits are syndrome testable for their internal lines and can be made syndrome testable for primary input faults through additional hardware in the design

It has been proved [18] that by using a weighted sum of syndromes of all the outputs, single stuck-at-faults, single bridging faults and single cross-point faults for all internal and input lines in any implementation of PLA are covered

4.1 Notations & Definitions

$\mathcal{F}(x_1, x_2, \quad, x_n)$ is a two-level and-or function of n inputs, represented in the list of cubes form as described in section 2 3 It is represented as, $\mathcal{F} = p_1 + p_2 + \quad + p_t$, (Fig 4 1)

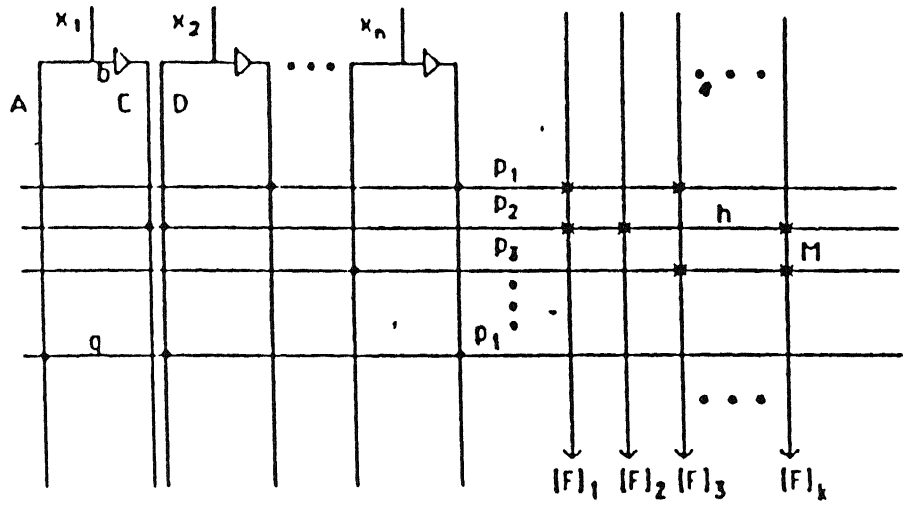


Figure 4 1 A PLA

where $x_1, x_2, \ldots, x_n$ are input lines, $[F]_1, [F]_2, \ldots, [F]_k$ are the output lines from the k output functions, $p_1, p_2, \ldots, p_t$ are product term lines

Definition 1 The *syndrome* $S(\mathcal{F})$ of a Boolean function $\mathcal{F}(x_1, x_2, \ldots, x_n)$ is defined as $S(\mathcal{F}) = W(\mathcal{F})/2^n$, where $W(\mathcal{F})$ is the weight of $\mathcal{F}$ and is given by the number of ON-vetices in the list of cubes representation of $\mathcal{F}$ The syndrome is a functional property Thus, various realizations of the same function will have the same syndrome

Example In the function $\mathcal{F}(x_1, x_2) = x_1 x_2$, the number of ON-vertices, i e , the no of true minterms is equal to 1, thus th syndrome $S(\mathcal{F}) = 1/4$ In general, the syndrome of an AND gate of n-inputs is $S = 2^{-n}$, for an OR gate of n-inputs $S = 1 - 2^{-n}$, for an EX-OR gate $S = 1/2$ etc

Definition 2 Given a set of k output functions $[\mathcal{F}]_1, [\mathcal{F}]_2, \ldots, [\mathcal{F}]_k$, define the weighted syndrome sum $WSS = \sum_{s=1}^{k} w_s W([\mathcal{F}]_s)$, where $w_1, w_2, \ldots, w_k$ are set of weights such that $w_s = 2^{s-1}, 1 \leq s \leq k$

Definition 3 If the function $\mathcal{F}$ is written as $\mathcal{F} = \bar{x}_i f_0 + x_i f_1$ (Shannon decomposition), the first order Walsh spectral coefficient r_i is given by the number of 1's in f_0 minus the number of 1's in f_1, i e , $r_i = W(f_0) - W(f_1)$

Definition 4 If $\mathcal{F}$ is written as

$$\mathcal{F} = \bar{x}_i \bar{x}_j f_{00} + \bar{x}_i x_j f_{01} + x_i \bar{x}_j f_{10} + x_i x_j f_{11}, \tag{4 1}$$

the second order spectral coefficient r_{ij} is given by $r_{ij} = W(f_{00}) + W(f_{01}) + W(f_{10}) + W(f_{11})$

Definition 5 For each first and second order coefficient r_i and r_{ij}, R_i and R_{ij} denote the corresponding weighted sum of the coefficients over all functions $[\mathcal{F}]_s$ as

$$R_i = \sum_{s=1}^{k} w_s [r_i]_s \tag{4 2}$$

$$R_{ij} = \sum_{s=1}^{k} w_s [r_{ij}]_s \tag{4 3}$$

for the set of weights w_s defined above, where $[r_i]_s$ and $[r_{ij}]_s$ are r_i and r_{ij} for output $[\mathcal{F}]_s$

4.2 The Classification of Faults in PLAs

There are three types of single faults to be considered

Stuck-at-faults One line, or line segment, stuck at either logical 1 or 0

Bridging faults Two adjacent lines are shorted together assuming the value of the logical AND or OR of their own individual signals (determined by the hardware implementation)

Cross-point faults The loss of contact of a connection or its spurious presence

Each type is considered separately

The internal single stuck-at-faults in a PLA may be grouped together as shown in Table 4 1 with explanatory remarks This is a detailed enumeration of types of faults Later these are grouped as to fault effects Each class is exemplified by referring to the layout of Fig 4 1 For clarity all vertical lines are labeled with capital letters, and horizontal lines with small letters

Example Considering type 1, which is a s-a-0 fault between literals, is the fault on the segment connecting two adjacent literals in the AND array, $e\ g$, the horizontal connection between the AND connections at x_1 and x_2(line g in Fig 4 1) Now, let us consider type 7, s-a-0 between products it is the fault on the segment connecting two adjacent product lines in the OR array, $\epsilon\ g$, the vertical segment between the OR connections of p_2 and p_3 (line M in Fig 4 1)

Cross-point faults, $i\ e$, erroneous connections between two crossing lines of the arrays, can be seen to be of four types

Fault type
1) s-a-0 between literals ($e\ g$, line $g/0$)
2) s-a-1 between literals ($e\ g$, line $g/1$)
3) s-a-0 sec input ($e\ g$, line $A/0$, line $C/0$, line $b/1$)
4) s-a-1 sec input ($e\ g$, line $A/1$, line $C/1$, line $b/0$)
5) s-a-0 product line ($e\ g$, line $p_1/0$, line $h/0$)
6) s-a-1 product line ($e\ g$, line $p_1/1$, line $h/1$)
7) s-a-0 between products ($e\ g$, line $M/0$)
8) s-a-1 between products ($e\ g$, line $M/1$)

Table 4 1 Types of faults illustrated with respect to Fig 4 1

Growth fault Loss of AND contact between product line and secondary input line, where one should exist ($e\ g$, no contact at p_2 and C in Fig 4 1)

Shrinkage fault AND contact between product line and secondary input line, where one should not be ($e\ g$, contact at p_1 and C in Fig 4 1)

Appearance fault OR contact between product line and output line, where one should not be ($e\ g$, contact at p_2 and $\mathcal{F}_3$ in Fig 4 1)

Disappearance fault Loss of OR contact between product line and output line, where one should exist ($e\ g$, no contact at p_1 and $\mathcal{F}_1$ in Fig 4 1)

A similar enumeration can be applied to the possible bridging faults

1 Bridging fault between two secondary inputs after inverter ($e\ g$, A and C in Fig 1 1)

2 Bridging fault between two product lines ($e\ g$, p_i and p_j in Fig 4 1)

3 Bridging fault between two output lines in OR array ($e\ g$, $\mathcal{F}_s$ and $\mathcal{F}_t$ in Fig 4.1)

4 Bridging fault between two primary inputs ($e\ g$, x_i and x_j in Fig 4 1)

The bridging will be either AND bridging or OR bridging depending on the technology

4.3 Testability of Stuck-at-faults

Any change in the syndrome of a function implies that the total number of minterms realised has increased or decreased Clearly, an increase or decreas in the syndrome of one output causes a corresponding change in the WSS

Fault type	Increase	Decrease
1) s-a-0 between literals		√
2) s-a-1 between literals	√	
3) s-a-0 sec input		√
4) s-a-1 sec input	√	
5) s-a-0 product line		√
6) s-a-1 product line	√	
7) s-a-0 between products		√
8) s-a-1 between products	√	

Table 4 2 Summary of s-a-f classes

A PLA implementation is based on sum-of-products-expressions(SOPE) Any given stuck-at-fault affects one or more product terms, but all are affected in an unidirectional manner It is demonstrated that any internal stuck-at-fault causes all the affected syndromes to increase or decrease, there being no possibility of cancellation

Suppose a literal is stuck at constant 1 or is disconnected from the AND line This is a growth fault [35], and all affected product terms loose a literal leading to an expansion in the number of minterms they cover Similarly, if a literal is incorrectly connected to an AND line(shrinkage fault), all affected product terms will gain that literal and there is a reduction in the number of covered minterms If a literal is stuck at constant 0, an entire product term disappears(disappearance fault) and WSS of the function will decrease

Example Let us consider a stuck-at-fault of class 1 of Table 4 1 If line g in Fig 4 1 is stuck-at 0, the whole product term p_t is ANDed to a result for all input configurations Thus, some minterms previously covered by p_t loose their coverage causing a decrease in the weight of allthose functons sharing p_t

Similarly, let a line A be stuck-at-1(class 4 of Table 4 1) Then all productterms connected to A will double in variable x_1 that is, thay cover minterms both for $x_1 = 1$ and $x_1 = 0$ Thus, an increase in the number of minterms covered happens for all functions dependent on x_1

Table 4 2 gives a summary of the stuck-at-fault classes as listed previously, with corresponding increase/decrease on the size of the product terms marked beside them

The examination of each type of fault demonstrates that a single fault cannot cause a literal to be both removed and added in the sum-of-products expressions, due to the structure of the PLA itself Thus the corresponding fault effect on the WSS is unidirectional

Hence, all internal single stuck-at-faults are covered by the WSS However, as will be seen in the next section all stuck-at-faults at the primary inputs are not WSS-testable

4.3.1 Primary Input Stuck-at-faults

If an input line is stuck-at-0 or stuck-at-1, both secondary input lines receive the effect of the fault, where on the negative line the fault is also inverted This is analogous to two separate faults, one of class 3 and one of class 4 (Table 4 1) From Table 4 2 it can be seen that one fault increases the individual syndromes while the other decreases them Thus, there is a possibility of cancellation

It is important to determine when and if this cancellation may occur in order to maintain WSS testability for the PLA A general result has been proved [18], which is given below

Theorem 1 *$x_i/0$ and $x_i/1$ are WSS testable if and only if $R_i \neq 0$, where $R_i = \sum_{s=1}^{k} w_s[r_i]_s$*

Proof Output $[\mathcal{F}]_s$ can be written as

$$[\mathcal{F}]_s = \bar{x}_i[f_0]_s + x_i[f_1]_s \tag{4 4}$$

Let $m_{0s} = W([f_0]_s), m_{1s} = W([f_1]_s), (1 \leq s \leq k)$ Now,

$$WSS = \sum_{s=1}^{k} w_s(m_{0s} + m_{1s}) \tag{4 5}$$

$$R_i = \sum_{s=1}^{k} w_s[r_i]_s \tag{4 6}$$

$$= \sum_{s=1}^{k} w_s(m_{0s} - m_{1s}) \tag{4 7}$$

Now, let us consider $x_i/0$ $W([\mathcal{F}]_s)^*$ (in the presence of the fault) is $2m_{0s}$ and $WSS^* = 2\sum_{s=1}^{k} w_s m_{0s}$, since $[f_0]_s$ is evaluated twice by the test procedure It follows that $WSS^* \neq WSS$ if and only if $R_i \neq 0$ Similarly, for $x_i/1$

Given this result, only each R_i need be checked if nonzero, then the corresponding input line is testable by the weighted sum of syndromes, and no cancellation occurs This verification that each R_i is nonzero is carried out at the design stage and it is not part of the actual test At testing time, only the WSS has to be calculated

Moreover, the spectral coefficients are such that, all of them are either odd or even for any given completely specified function $[\mathcal{F}]_s$ This is because of the orthogonality of

the transformation matrix T [2] For the set of weights w_s, it is clear that $R_i = [r_i]_1 + 2A$ for some A, since the weights are multiples of 2 (except w_1) Thus $R - i$ is even or odd according as $[r_i]_1$ is even or odd, this being determined by whether $\mathcal{F}_1$ is of even or odd weight However the ordering of output functions can be permutated to ensure that a function of odd weight becomes the right most output, i e , $\mathcal{F}_1$ This selection is done at the design time

For an incompletely specified function, in the PLA implementation the don't care product terms are not implemented Thus it becomes equivalent to a completely specified function, with its DC-cubes converted to OFF-cubes and the above considerations are valid

The only time when the above permutation method fails is when all functions being realized are of even weight After calculating R_i in such a case, and finding it equal to 0, the simplest hardware change done is the addition of an extra output line This extra function realized in the PLA should have only one minterm, $y_1 y_2 \quad y_n \in \{x_i, \vec{x}_i\}, 1 \leq i \leq n$ This extra output is made $[\mathcal{F}]_1$, such that its w_1 is 1 In this way the value of R_i becomes $+1$ or -1, and the PLA becomes WSS testable for primary input stuck-at-faults

4.4 Testability of Cross-point Faults

The four types of cross-point faults are covered in a similar manner to the stuck-at-faults

A shrinkage fault between a product line p_s and a secondary input line x_i causes the coverage of p_s to either be halved, if it is not already connected to $\vec{x}_i$, or to go to zero, if it is already connected to $\vec{x}_i$ Thus the weights of all functions involving p_s are either decreased The same applies for a contact between p_s and $\vec{x}_i$

Conversely, for a growth-fault causes the p_s involved to become independent of the variable x_i Thus, the product term doubles in size along x_i and the weights for all the functions sharing it increase

For an appearance-fault, an extra product term is realised for the function involved causing an increase in the number of minterms covered (note that there will be no change if there is redundancy) A disappearance-fault for a function $[\mathcal{F}]_s$ reduces the number of minterms covered

Thus, all internal single cross-point faults are covered by WSS

4.5 Testability of Bridging Faults

The five classes for AND bridging faults are considered here in detail Similar results follow for OR bridging faults

1 *Bridge between x_i and $\vec{x}_i$* Any output $[\mathcal{F}]_s$ that depends on x_i can be written as,

$$[\mathcal{F}]_s = Ax_i + B\vec{x}_i + C, \tag{4 8}$$

for some A, B, C Then under the AND bridging fault between x_i and $\vec{x}_i$, it becomes

$$\begin{aligned} [\mathcal{F}]_s^* &= A(x_i\vec{x}_i) + B(x_i\vec{x}_i) + C \\ &= C \end{aligned} \tag{4 9}$$

Clearly $W[\mathcal{F}]_s$ is less than the fault-free case and WSS changes accordingly

2 *Bridge between $\vec{x}_i$ and x_j* This fault causes both $\vec{x}_i$ and x_j to be replaced by $(\vec{x}_i x_j)$ Then,

$$[\mathcal{F}]_s = Ax_ix_j + Bx_i\vec{x}_j + C\vec{x}_ix_j + D\vec{x}_i\vec{x}_j + E \tag{4 10}$$

for some A, B, C, D, E changes to,

$$[\mathcal{F}]_s^* = B(x_i\vec{x}_j) + C(\vec{x}_ix_j) + E, \tag{4 11}$$

with a corresponding decrease in $W([calF]_s)$ and WSS

3 *Bridge between product term lines p_i and p_j* This fault causes both the product terms to be replaced by (p_ip_j) An output $[\mathcal{F}]_s = p_1 + p_2 + \quad + p_i + \quad + p_j +$ becomes

$$[\mathcal{F}]_s^* = p_1 + p_2 + \quad . + p_ip_j + \quad , \tag{4 12}$$

with a resulting decrease in weight

4 *Bridge between output lines $[\mathcal{F}]_s$ and $[\mathcal{F}]_t$* This type of fault in the OR array causes the two outputs to become

$$[\mathcal{F}]_s^* = [\mathcal{F}]_t^* = [\mathcal{F}]_s[\mathcal{F}]_t \tag{4 13}$$

with a decrease in weights of both the functions and WSS

5 *Bridge between primary inputs x_i and x_j* This is a special case and we consider it in the next section

4.5.1 Primary Input Bridging Faults

In the case of an AND bridging fault between two primary inputs, x_i, x_j, the effect is carried to both secondary input lines with the following multiple substitutions taking place

- each of x_i and x_j is replaced by $x_i x_j$
- each of $\bar{x}_i$ and $\bar{x}_j$ is replaced by $\bar{x}_i \bar{x}_j$

If $[\mathcal{F}]_s$ is written as Equation 4.10, then under the fault it becomes,

$$[\mathcal{F}]_s = Ax_i x_j + D\bar{x}_i + D\bar{x}_j + E, \tag{4.14}$$

and a cancellation potential exists, since there is no guarantee that the number of minterms uniformly decreases or increases.

To isolate this possibility, a condition similar to the one previously discussed for stuck-at faults on primary inputs can be proved. If $\mathcal{F}$ is expressed as in Equation 4.1, for an AND bridging fault between two primary inputs, the subfunctions f_{01} and f_{10} assume the same value as f_{00}. Thus no cancellation can occur if and only if

$$W(f_{01}) + W(f_{10}) \neq 2W(f_{00}) \tag{4.15}$$

This condition can be expressed in the spectral domain using the first and second order spectral coefficients, and here the general result is proved [18].

Theorem 2 *An AND bridging fault between two primary inputs x_i and x_j is WSS testable if and only if $R_i + R_j + R_{ij} \neq 0$.*

Proof We have,

$$[\mathcal{F}]_s = \bar{x}_i \bar{x}_j [f_{00}]_s + \bar{x}_i x_j [f_{01}]_s + x_i \bar{x}_j [f_{10}]_s + x_i x_j [f_{11}]_s \tag{4.16}$$

Let $m_{0s} = W([f_{00}]_s), m_{1s} = W([f_{01}]_s), m_{2s} = W([f_{10}]_s), m_{3s} = W([f_{11}]_s), (1 \leq s \leq k), k =$ number of outputs. Now,

$$WSS = \sum_{s=1}^{k} w_s(m_{0s} + m_{1s} + m_{2s} + m_{3s}) \tag{4.17}$$

$$R_i = \sum_{s=1}^{k} w_s[r_i]_s \tag{4.18}$$

$$= \sum_{s=1}^{k} w_s(m_{0s} + m_{1s} - m_{2s} - m_{3s}) \tag{4.19}$$

$$R_j = \sum_{s=1}^{k} w_s[r_j]_s \quad (4\ 20)$$

$$= \sum_{s=1}^{k} w_s(m_{0s} - m_{1s} + m_{2s} - m_{3s}) \quad (4\ 21)$$

$$R_{ij} = \sum_{s=1}^{k} w_s[r_{ij}]_s \quad (4\ 22)$$

$$= \sum_{s=1}^{k} w_s(m_{0s} - m_{1s} - m_{2s} + m_{3s}) \quad (4\ 23)$$

With AND bridging between x_i and x_j, the weights become

$$W^*([f_{01}]_s) = W^*([f_{10}]_s) = W([f_{00}]_s) = m_{0s}$$

Thus, $WSS = \sum_{s=1}^{k} w_s(3m_{0s} + m_{3s})$

It follows that $WSS^* \neq WSS$ if and only if $R_i + R_j + 2R_{ij} \neq 0$ Similarly, for an OR bridging fault, it can be proved [18] that bridging fault between primary inputs x_i and x_j is testable if $R_i + R_j - 2R_{ij} \neq 0$

Thus at the design stage the R_i, R_j and R_{ij} for each pair of physically adjacent input lines (there are $n-1$ such lines), should be checked If the calculated sum is 0, for any i and j, then the same change in hardware as suggested in the case of stuck-at-faults, *i e*, addition of an extra output line works effectively

In the case of AND bridging faults note that the extra minterm, previously written as $y_1 y_2 \quad y_n$, where each y_i is a literal being either x_i or $\bar{x}_i$, is best implemented as $\bar{x}_1 \bar{x}_2 \quad \bar{x}_n$ This particular product term will affect only the total weight of the f_{00} decomposed portions of the output functions by adding one to it and leaving the other two total weights, *i e*, $W(f_{01})$ and $W(f_{10})$ unaltered This remains true for any pair of (x_i, x_j) since, it is always just $W(f_{00})$ that is affected

This technique is described in the algorithm below In the first step the PLA is expressed as a list of disjoint ON-cubes for each output with the algorithm of section 2 3 Then, the calculation of WSS becomes easy, since weight of each function is the total number of vertices in all the cubes in the list The number of vertices in a cube C is given by, $W(C) = 2^d$, where d = number of don't cares in C Also the same list is used to calculate the *Ist* and *2nd* order spectral coefficients using the algorithm described in section 2 4 Then it recursively checks for the s-a-f and bridging fault testability at the primary inputs If, either for a bridging fault or for both a s-a-f and a bridging fault at the primary

inputs, the PLA is found untestable, then an extra output line is added, which implements only the 0-th minterm However, if it is untestable for only a s-a-f at primary inputs, the output lines are reordered so that $[\mathcal{F}]_1$ corresponds to an output of odd weight making the corresponding $s_i \neq 0$

Algorithm Checks the WSS testability of a PLA. If found untestable, modifies it to make it WSS-testable

```
for each output,
        express the PLA as a list of disjoint ON-cubes,
bdg = saf = 0,
do {
    for each output i = 1 until k
            calculate the Ist and 2nd order spectral coefficients
    for each i = 1 until n, /* n = no of inputs */
            if (R_i == 0)saf = 1,
    for each i = 1 until n - 1 /* Check for each adjacent pairs of
                                  primary inputs */
            if ((R_i + R_{i+1} + 2R_iR_{i+1}) == 0)bdg = 1,
    if (bdg == 1)
            if product term (x_i = 0, 1 <= i <= n) is present in the PLA
                    add an extra output [F]_1 having only this product term,
            else
                    add the 0-th minterm as a product term in the PLA,
                    add an extra output [F]_1 having only the 0-th minterm,
    else if (saf == 1)
            selectflag = 0,
            do {
                select an output line r randomly,
                if(W(F_r) is odd) selectflag = 1,
            }while (selectflag ≠ 1),
            interchange the outputs [F]_r and [F]_1
    } while (saf == 1 or bdg == 1),
calculate the WSS of the PLA,
```

4.6 Coverage of Multiple faults

The assumption of single faults of any type may not be sufficient for full testability However, it may be reasonable to expect that, a set of faults occuring simultaneously are not totally random in their appearance and distribution The testability criteria as discussed in the earlier sections of this chapter for single faults by WSS are based on the unidirectionality of the change in the syndromes Thus it follows that any combination of faults having the same unidirectional effect is fully covered by WSS

The result acquires more significance by noticing that the most common faults come

from the error in programming the PLA, resulting in too many or too few connections, or in shorting of a set of adjacent lines Hence the WSS is a strong testability condition for the largest and most common set of multiple faults, which indeed lead to unidirectional effects [18]

4.7 Implementation and Results

This algorithm has been implemented as a computer program, "SPECTEST", written in 'C' It has the option of calculating the WSS only, checking for complete WSS-testability and making the necessary hardware modification, and finding the fault in a faulty PLA if the fault-free PLA or its syndrome is given as input It has been tested on a set of MCNC PLA benchmarks, and some commonly used functions The results are shown in Table 4 3

	Testability			*Testing*
Example	*S-a-f*	*Bridging*	*WSS*	*time(s)*
5xp1	√	√	55961	0 56
Z5xp1	√	√	40768	1 42
9sym	×	√	849 ⋆	2 17
Z9sym	×	√	841 ⋆	4 63
rd53	√	√	76	0 24
rd73	√	√	449	0 98
rd84	√	√	1636	2 79
sao2	√	√	1417	1 03
add2	√	√	48	0 11
add3	√	√	448	0 34
add4	√	√	3840	2 98
mult3	√	√	784	0 28
mult4	√	√	14400	2 93

Table 4 3 Table of test results of PLA benchmarks √ implies testability and × implies untestability ⋆ implies it is the WSS of the modified PLA The times are given for s-a-0 faults at primary inputs

Chapter 5

Conclusion and Future Work

The essential relationships between classical and spectral methods used in the design of digital circuits have been stated Based on these relations, algorithms for generation of spectral coefficients for both **S** and **R** spectra for completely as well as incompletely specified Boolean functions have been deshown Such detailed interpretations of **S** and **R** spectra are important not only from the point of view analysis and synthesis of digital systems, but also for the generation of test sets and design for testability

Linear decomposition, described in chapter 3, has no equivalent Boolean domain solutions [5] It is a good supplement to other synthesis techniques because it reduces the logic complexity of many functions that are usually difficult to synthesize efficiently in the Boolean domain It has also been found to improve testability of implementations because EX-OR networks are known to be easily testable for single stuck-at-faults Linearized implementations are also known to be syndrome testable, which makes them suitable for built-in test

The weighted syndrome sum for all outputs in a PLA described in chapter 4 is a sufficient test to cover single stuck-at-faults, bridging faults and cross point faults The model adopted did not explicitly consider all multiple faults However, all multiple faults belonging to classes possessing the same direction of change in their WSS are fully cobered The weighted sum of syndromes is a very simple yet very powerful coding for the testing of PLAs and its straightforward implementation makes it suitable both for easy design for testability (DFT) and built in self test (BIST)

Future work in this area can be development of new decomposition methods for systems of incompletely specified Boolean functions based on the representation of the Hadamard-Walsh spectrum presented The properties of such decompositions make them very suitable for design using Field Programmable Gate Arrays (FPGAs) Also, some work

on the area of multiple fault detection using the syndrome technique can be taken

Bibliography

[1] C R Edwards, "The application of the Rademacher–Walsh transform to Boolean function classification and threshold logic synthesis," *IEEE Trans Comput*, vol C-24, pp 48-62, Jan 1975

[2] S L Hurst, D M Miller, and J C Muzio, *Spectral Techniques in Digital logic* London, U K Academic, 1985

[3] J C Muzio, "Composite spectra and the analysis of switching circuits," *IEEE Trans Comput*, vol C-29, pp 750-753, 1980

[4] J C Muzio, D M Miller, and S L Hurst, "Number of spectral coefficients necessary to identify a class of Boolean functions," *Electron Lett*, vol 25, pp 577-578, 1982

[5] D Varma and E A Trachtenberg, "Design automation tools for efficient implementation of logic functions by decomposition," *IEEE Trans Computer-Aided Design*, vol 8, pp 901-916, Aug 1989

[6] E Eris and J C Muzio, "Spectral testing of circuit realisations based on linearizations," *Proc IEE Comput & Digital Tech*, vol 133, pp 73-78, 1986

[7] M G Karpovsky, *Finite Orthogonal Series in Design of Digital Devices* New York Wiley, 1976

[8] M G Karpovsky, ed , *Spectral Techniques and Fault Detection* Orlando, FL, Academic, 1985

[9] A M Lloyd, "Design of multiplexer universal-logic-module networks using spectral techniques," *Proc IEE Comput & Digital Tech*, vol 127, pp 31-36, 1980

[10] R Lechner, "Harmonic analysis of switching functions," in *Recent Developments in Switching Theory* A Mukhopadhyay, Ed , pp 121-128, New York Academic, 1971

[11] D Varma and E A Trachtenberg, "A fast algorithm for the optimal state assignment of large finite state machines," *Proc IEEE Int Conf on Computer-Aided Design*, Santa Clara, CA, pp 152-155, 1988

[12] T Hsiao and S C Seth, "An analysis of the use of Rademacher–Walsh spectrum in compact testing," *IEEE Trans Comput*, vol C-33, pp 934-938, Oct 1984

[13] S L Hurst, "Use of linearization and spectral techniques in input and output compaction testing of digital networks," *Proc IEE Comput & Digital Tech*, vol 136, pp 48-56, Jan 1989

[14] P K Lui and J C Muzio, "Spectral signature testing of multiple stuck-at faults in irredundant combinatorial networks," *IEEE Trans Comput*, vol C-35, pp 1088-1092, Dec 1986

[15] D M Miller and J C Muzio, "Spectral fault signatures for single stuck-at faults in combinatorial networks," *IEEE Trans Comput*, vol C-33, pp 765-768, Aug 1984

[16] D M Miller, ed , *Developments in Integrated Circuit Testing* London, UK Academic, 1987

[17] J C Muzio and D M Miller, "Spectral fault signatures for internally unate combinatorial networks," *IEEE Trans Comput*, vol C-32, pp 1058-1062, 1983

[18] M Serra and J C Muzio, "Testing Programmable Logic Arrays by sum of syndromes," *IEEE Trans Comput*, vol C-36, pp 1097-1101, Sept 1987

[19] M Serra and J C Muzio, "Space compaction for multi-output circuits," *IEEE Trans Computer-Aided Design*, vol 7, pp. 1105-1113, Oct 1988

[20] R K Brayton, G D Hachtel, C T. McMullen, and A L Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Hingham, MA Kluwer Academic, 1985

[21] J M Sanchez, J Ballesteros, and A Vaquero, "Study of the complexity of an algorithm to derive the complement of a binary function," *Int J Electron*, vol 66, pp 245-250, 1989

[22] Ph W Besslich "Spectral processing of switching functions using signal flow transformations," in *Spectral Techniques and Fault Detection* M G Karpovsky, Ed , Orlando, FL Academic, 1985

[23] B J Falkowski, I Schäfer, and M A Perkowski, "Effective computer methods for the calculation of Rademacher-Walsh spectrum for completely and incompletely specified boolean functions," *IEEE Trans Computer-Aided Design*, vol 11, pp 1207-1226, Oct 1992

[24] B J Falkowski and M A Perkowski, "Algorithm for the generation of disjoint cubes for completely and incompletely specified boolean functions," *Int J Electron* , vol 70, pp 533-538, 1991

[25] B J Falkowski and M A Perkowski "One more method for the calculation of Hadamard Walsh spectrum for completely and incompletely specified boolean functions," *Int J Electron* , vol 69, pp 595-602, Nov , 1990

[26] J P Roth, *Computer Logic, Testing and Verification* Potomac, MD Computer Science Press, 1980

[27] M Helliwell and M A Perkowski, "A fast algorithm to minimize multi-output mixed-polarity generalized Reed Muller forms," *Proc 25th ACM/IEEE Design Automation Conf* , Anaheim, CA, pp 427-432, Jun 1988

[28] L Nguyen M A Perkowski and N Goldstein, "PALMINI–fast Boolean minimizer for personal computers," *Proc 24th ACM/IEEE Design Automation Conf* , pp 615-621, 1987

[29] J C Muzio and S L Hurst, "The computation of complete and reduced sets of orthogonal spectral coefficients for logic design and pattern recognition purposes," *Comput & Elect Eng* , vol 5, pp 231-249, 1978

[30] T Sasao, "Input variable assignment and output phase optimization of PLAs," *IEEE Trans Comput* vol C 33, pp 879-894, Oct 1984

[31] G Rajagopalan "Design of VLSI modules using decoded PLA," *M Tech Thesis*, EE Dept , IIT Kanpur, Apr 1989

[32] T Sasao, "Multiple-valued logic and optimization of programmable logic arrays," *COMPUTER*, pp. 71-80, Apr 1988.

[33] R Lechner and A Moezzi, "The synthesis of encoded programmable logic arrays," in *Spectral Techniques and Fault Detection*, M Karpovsky, Ed , Orlando, FL. Academic, 1985.

[34] R K. Brayton, R Rudell, A L Sangiovanni-Vincentelli, and A R Wang, "MIS A multiple-level logic optimization system," *IEEE Trans Computer-Aided Design*, vol CAD-6, no 6, pp 1062-1081, Nov 1987

[35] J E Smith, "Detection of faults in programmable logic arrays," *IEEE Trans Comput.*, vol C-28, pp. 845-853, Nov 1979

[36] J Savir, "Syndrome-testable design of combinational circuits," *IEEE Trans Comput.*, vol C-29, pp 442-451, Jun 1980